```python
"""
Copyright (c) 2018 Grant Perkins
"""

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from time import time
from depthview import Dataset
import math


class SparseAutoEncoder:
    """
    SparseAutoEncoder is an implementation of a sparse autoencoder for the MNIST dataset. This implementation
    defines an encoding function that reduces an image to its primitive features. It is reduced from 28x28 to 10x10.
    """

    def __init__(self, input_size=4800, hidden_size=4800, rho=0.01, theta=.0001, beta=3):
        """
        Sets all constants and initializes weights and biases
        :param input_size: Size of input and output layer (1D)
        :param hidden_size: Size of hidden layer (1D)
        :param rho: Desired average sparsity
        :param theta: Weight decay parameter, actually lambda in papers
        :param beta: Weight of sparsity penalty term
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.rho = rho
        self.theta = theta
        self.beta = beta

        self.optimizer = tf.train.AdamOptimizer(learning_rate=1e-4)

        with tf.variable_scope("Variables"):
            self.W1 = self._new_variable([self.input_size, self.hidden_size], "Weight1")
            self.b1 = self._new_variable([1, self.hidden_size], "Bias1")

            self.W2 = self._new_variable([self.hidden_size, self.input_size], "Weight2")
            self.b2 = self._new_variable([1, self.input_size], "Bias2")

        self.sess = tf.Session()
        self.train_writer = tf.summary.FileWriter("saefilewriter/")
        self.saver = tf.train.Saver(name="Save")

    def _new_variable(self, shape, name):
        """
        Initializes a variable of given shape with random contents
        :param shape: Shape of the tensor
        :return: a tf.Variable of given shape, random contents
        """
        values = tf.random_normal(shape, stddev=.05)
        return tf.Variable(values, name=name)

    def _encode(self, X):
        """
        Encodes data X into a smaller layer
        :param X: images as a tensor
        :return: encoded layer
        """
        with tf.variable_scope("Encode"):
            return tf.nn.sigmoid(tf.matmul(X, self.W1) + self.b1, name="sigmoid")
```

```
64
65      def _decode(self, H):
66          """
67          Decodes encoded data H into a larger layer
68          :param H: encoded images as tensor
69          :return: decoded layer
70          """
71          with tf.variable_scope("Decode"):
72              return tf.nn.sigmoid(tf.matmul(H, self.W2) + self.b2, name="sigmoid")
73
74      def _kl_divergence(self, rho_hat):
75          """
76          Computes the Kullback-Leibler divergence
77          Divergence is between rho and rho hat
78          :param rho_hat: average activation of all nodes in encoding layer
79          :return: KL divergence value
80          """
81          with tf.variable_scope("KLDivergence"):
82              return self.rho * (tf.log(self.rho+1e-10) - tf.log(rho_hat+1e-10)) + (1 - self
    .rho) * (tf.log((1 - self.rho+1e-10)) - tf.log(1 - rho_hat+1e-10))
83
84      def _cost(self, X):
85          """
86          Computes current cost of the autoencoder
87          :param X: input images
88          :return: current cost
89          """
90          with tf.variable_scope("Cost"):
91              H = self._encode(X)
92              X_hat = self._decode(H)
93
94              with tf.variable_scope("Loss"):
95                  diff = X - X_hat
96                  rho_hat = tf.reduce_mean(H, axis=0, name="rho_hat")
97                  kl = self._kl_divergence(rho_hat)
98                  cost = .5 * tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1)) \
99                         + .5 * self.theta * (tf.reduce_sum(self.W1 ** 2) + tf.reduce_sum(
    self.W2 ** 2)) \
100                        + self.beta * tf.reduce_sum(kl)
101                 return cost
102
103     def train(self, data, train_steps=100):
104         """
105         Trains autoencoder
106         :param data: all input images
107         :param train_steps: amount of training steps.
108         :return: None
109         """
110         print("Training started")
111         start = time()
112         batch_size = 10
113
114         print("Cropping images")
115         images = data.frames
116         images = tf.convert_to_tensor(images)
117         print(images.shape)
118         print("All concatenated")
119         images = tf.image.resize_images(images, tf.convert_to_tensor([60, 80]))
120         print("Images cropped to ", images.shape)
121         images=self.sess.run(images).reshape(images.shape[0], 4800)
122
123         X = tf.placeholder(tf.float32, shape=[None, 60 * 80], name="X")
124         cost = self._cost(X)
125         optimizer = self.optimizer.minimize(cost)
126         self.sess.run(tf.global_variables_initializer())
```

```
        self.train_writer.add_graph(self.sess.graph)
        for step in range(train_steps):
            if step % 5 == 0:
                data.restart()
            for batch in range(100):
                x_batch = data.next_batch(batch_size, images)
                self.sess.run(optimizer, feed_dict={X: x_batch})
            print("Seconds since training started:", time() - start, "Step", step)
        save_path = self.saver.save(self.sess, "/var/sae.cpkt")
        print("Model saved to", save_path)

    def save_weight_picture(self, plots_len=4, file_name="trained_weights_2.png"):
        """
        Saves some of the trained weights of the encoding layer as an image
        :param image_len: length of a square image
        :param plots_len: length of the square grid of plots
        :param file_name: name of output file
        :return: None
        """
        images = self.W1.eval(self.sess)
        images = images.transpose()

        figure, axes = plt.subplots(nrows=plots_len, ncols=plots_len)
        for i, axis in enumerate(axes.flat):
            axis.imshow(images[i+500, :].reshape(60, 80), cmap=plt.get_cmap("binary"))
            axis.set_axis_off()
        plt.savefig(file_name)
        print("Picture of weights saved to", file_name)
        plt.imshow(images[476, :].reshape(60, 80), cmap=plt.get_cmap("binary"))

        file_name=file_name[:-4]+"2"+".png"
        plt.savefig(file_name)
        print("Picture of weights saved to", file_name)
        plt.close()

    def encode(self, image):
        """
        Encodes image
        :param image: a tensor containing an image
        :return: encoded image
        """
        return self.sess.run(self._encode(image))


def main():
    """
    Creates and trains sparse autoencoder
    :return: None
    """
    #data = Dataset()
    sae = SparseAutoEncoder()
    #sae.train(data)
    sae.saver.restore(sae.sess, "/var/sae.cpkt")
    sae.save_weight_picture(2)


if __name__ == '__main__':
    main()
```